Programación Funcional Avanzada

Zippers

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"

Transformando estructuras puras

- Iteraciones implícitas sobre una estructura.
  - `map` – modificar *todos* los contenidos preservando la estructura.
  - `fold` – recorrer *toda* la estructura produciendo una nueva.

UNIVERSIDAD SIMÓN BOLÍVAR

Transformando estructuras puras

- Iteraciones implícitas sobre una estructura.
  - `map` – modificar *todos* los contenidos preservando la estructura.
  - `fold` – recorrer *toda* la estructura produciendo una nueva.
- ¿Qué hacer cuando se necesitan cambios selectivos?
  - Cambiar **un** elemento de una lista, **un** nodo de un árbol, ...
  - Y si ese cambio implica *otro* cambio selectivo en la misma estructura, ¿cómo proseguir?

Recorriendo listas

La estrategia convencional

- Si tenemos una lista de valores...

```
let lista = [4,17,8,25,19,42,17,3,18]
```

- ...y queremos cambiar el 42 por 69, escribimos algo como ...

```
r42f69 :: [Int] -> [Int]
r42f69 []     = []
r42f69 (x:xs) = if x == 42 then (69 : xs)
                           else (x  : r42f69 xs)
```

UNIVERSIDAD SIMÓN BOLÍVAR

Recorriendo listas

La estrategia convencional

- Si tenemos una lista de valores...

```
let lista = [4,17,8,25,19,42,17,3,18]
```

- ... y queremos cambiar el 42 por 69, escribimos algo como ...

```
r42f69 :: [Int] -> [Int]
r42f69 []      = []
r42f69 (x:xs) = if x == 42 then (69 : xs)
                           else (x  : r42f69 xs)
```

- ... y como gran cosa la "generalizamos"

```
rep a b []     = []
rep a b (x:xs) = if x == a then (b : xs)
                           else (x : rep a b xs)
```

UNIVERSIDAD SIMÓN BOLÍVAR

Recorriendo listas

La estrategia convencional

- Si tenemos una lista de valores...

```
let lista = [4,17,8,25,19,42,17,3,18]
```

- ...y queremos cambiar el 42 por 69, escribimos algo como ...

```
r42f69 :: [Int] -> [Int]
r42f69 []     = []
r42f69 (x:xs) = if x == 42 then (69 : xs)
                           else (x  : r42f69 xs)
```

- ...y como gran cosa la "generalizamos"

```
rep a b []     = []
rep a b (x:xs) = if x == a then (b : xs)
                           else (x : rep a b xs)
```

Pero para cambiar otro valor,
hay que reprocesar desde el principio.

UNIVERSIDAD SIMÓN BOLÍVAR

Recorriendo árboles

Binarios con valores en los nodos internos

```
data Tree a = Leaf | Branch a (Tree a) (Tree a)
     deriving (Eq,Show)

testTree :: Tree Char
testTree =
    Branch 'A'
        (Branch 'R'
           (Branch 'P' Leaf Leaf)
           (Branch 'E' Leaf Leaf)
        )
        (Branch 'E'
           (Branch 'R' Leaf Leaf)
           (Branch 'A' Leaf Leaf)
        )
```

Cambiemos la 'P' por una 'N'

La solución directa y horrorosa

Podemos usar *pattern-matching* para hacer el cambio –
pero necesitamos un punto de referencia para avanzar.

```
r2n :: Tree Char -> Tree Char
r2n (Branch a (Branch r (Branch  _  m n) e) p) =
    (Branch a (Branch r (Branch 'N' m n) e) p)
```

UNIVERSIDAD SIMÓN BOLÍVAR

La solución directa y horrorosa

Podemos usar *pattern-matching* para hacer el cambio –
pero necesitamos un punto de referencia para avanzar.

```
r2n :: Tree Char -> Tree Char
r2n (Branch a (Branch r (Branch  _  m n) e) p) =
    (Branch a (Branch r (Branch 'N' m n) e) p)
```

- Funciona, pero es definitivamente feo y muy confuso – ¿pueden comprenderlo *sin* dibujar el árbol y seguir el paso a paso?
- ¿Y cómo lo generalizamos para cualquier nodo en cualquier árbol? – combinar algoritmo de búsqueda con deconstrucción y reconstrucción.

Abstraer el movimiento

. . . mientras reconstruimos

- Supongamos que no tenemos que buscar sino que sabemos llegar – contamos con el camino para alcanzar el nodo a cambiar.

```
data Direction  = L | R deriving (Show,Eq)
type Directions = [Direction]
```

  - En cada nodo escogemos el subárbol izquierdo (`L`) o derecho `D`).
  - Y tenemos una lista de pasos desde la raíz hasta el elemento – pasos que *nunca* nos hacen "caer del árbol".

UNIVERSIDAD SIMÓN BOLÍVAR

Abstraer el movimiento

. . . mientras reconstruimos

- Supongamos que no tenemos que buscar sino que sabemos llegar – contamos con el camino para alcanzar el nodo a cambiar.

```
data Direction  = L | R deriving (Show,Eq)
type Directions = [Direction]
```

  - En cada nodo escogemos el subárbol izquierdo (`L`) o derecho `D`).
  - Y tenemos una lista de pasos desde la raíz hasta el elemento – pasos que *nunca* nos hacen "caer del árbol".

- Eso lo hace mucho más fácil de implantar

```
r2n :: Directions -> Tree Char -> Tree Char
r2n (L:ds) (Branch x l r) = Branch x (r2n ds l) r
r2n (R:ds) (Branch x l r) = Branch x l (r2n ds r)
r2n []     (Branch _ l r) = Branch 'N' l r
```

Abstraer el movimiento

. . . sólo para recorrer

- Y si sólo queremos saber qué hay en una posición particular

```
elemAt :: Directions -> Tree a -> a
elemAt (L:ds) (Branch _ l _) = elemAt ds l
elemAt (R:ds) (Branch _ _ r) = elemAt ds r
elemAt []     (Branch x _ _) = x
```

UNIVERSIDAD SIMÓN BOLÍVAR

Abstraer el movimiento

. . . sólo para recorrer

- Y si sólo queremos saber qué hay en una posición particular

```
elemAt :: Directions -> Tree a -> a
elemAt (L:ds) (Branch _ l _) = elemAt ds l
elemAt (R:ds) (Branch _ _ r) = elemAt ds r
elemAt []     (Branch x _ _) = x
```

- La lista de movimientos permite **enfocar** el subárbol de interés – por eso la literatura habla de *lenses* para esta técnica.
- Estamos mejor, pero aún es ineficiente hacer cambios repetitivos – siempre tenemos que comenzar desde la raíz.

UNIVERSIDAD SIMÓN BOLÍVAR

Recordar nuestro avance

The Hansel and Gretel Technique®

- `Directions` indica los pasos que debemos seguir para avanzar – `Breadcrumbs` nos indicará cómo *regresar*.

```
type Breadcrumbs = [Direction]
```

UNIVERSIDAD SIMÓN BOLÍVAR

Recordar nuestro avance

The Hansel and Gretel Technique®

- `Directions` indica los pasos que debemos seguir para avanzar – `Breadcrumbs` nos indicará cómo *regresar*.

```
type Breadcrumbs = [Direction]
```

- Si avanzamos hacia la izquierda...

```
goLeft :: (Tree a, Breadcrumbs)
       -> (Tree a, Breadcrumbs)
goLeft (Branch _ l _, bs) = (l, L:bs)
```

- ...o hacia la derecha...

```
goRight :: (Tree a, Breadcrumbs)
        -> (Tree a, Breadcrumbs)
goRight (Branch _ _ r, bs) = (r, R:bs)
```

UNIVERSIDAD SIMÓN BOLÍVAR

Get lost in the woods

```
> goLeft $ goRight (testTree,[])
((Branch 'E' Leaf Leaf),[L,R])
> goLeft $ goLeft (testTree,[])
((Branch 'P' Leaf Leaf),[L,L])
```

- Mantenemos el “foco” en el subárbol de interés.
- Mantenemos el camino de regreso – noten el orden.

¿Cómo retroceder sobre nuestros pasos?

UNIVERSIDAD SIMÓN BOLÍVAR

¡Migas más grandes!

. . . que permitan reconstruir el nodo padre

- El rastro solamente contiene el camino de regreso
  - Necesitamos saber el árbol padre que dejamos atrás.
  - Y el "otro" camino que no tomamos.

¡Migas más grandes!

. . . que permitan reconstruir el nodo padre

- El rastro solamente contiene el camino de regreso
  - Necesitamos saber el árbol padre que dejamos atrás.
  - Y el "otro" camino que no tomamos.
- Mejoremos el rastro

```
data Crumb a = LeftCrumb a (Tree a)
             | RightCrumb a (Tree a)
               deriving (Eq,Show)

type Breadcrumbs a = [Crumb a]
```

  - En lugar de simples `L` o `R` tenemos una estructura.
  - Contiene el item en el nodo desde el que nos acabamos de mover.
  - Contiene el sub-árbol que *no* visitamos.

UNIVERSIDAD SIMÓN BOLÍVAR

Agreguemos la funcionalidad al movimiento

- Modificamos `goLeft` y `goRight` para usar las migas más grandes...

```
goLeft :: (Tree a, Breadcrumbs a)
       -> (Tree a, Breadcrumbs a)
goLeft (Branch x l r, bs) = (l, LeftCrumb x r:bs)

goRight :: (Tree a, Breadcrumbs a)
        -> (Tree a, Breadcrumbs a)
goRight (Branch x l r, bs) = (r, RightCrumb x l:bs)
```

UNIVERSIDAD SIMÓN BOLÍVAR

Agreguemos la funcionalidad al movimiento

- Modificamos `goLeft` y `goRight` para usar las migas más grandes...

```
goLeft :: (Tree a, Breadcrumbs a)
       -> (Tree a, Breadcrumbs a)
goLeft (Branch x l r, bs) = (l, LeftCrumb x r:bs)

goRight :: (Tree a, Breadcrumbs a)
        -> (Tree a, Breadcrumbs a)
goRight (Branch x l r, bs) = (r, RightCrumb x l:bs)
```

- ¡...y ahora podemos regresar por donde vinimos!

```
goBack :: (Tree a, Breadcrumbs a)
       -> (Tree a, Breadcrumbs a)
goBack (t, LeftCrumb  x r:bs) = (Branch x t r, bs)
goBack (t, RightCrumb x l:bs) = (Branch x l t, bs)
```

Y eso es un zipper

```
type Zipper a = (Tree a, Breadcrumbs a)
```

- Un par que contiene el foco y el rastro.
- Foco sobre la raíz de la subestructura – para trabajar o movernos.
- Rastro incluye movimiento y resto de la estructura que dejamos atrás.

UNIVERSIDAD SIMÓN BOLÍVAR

Sacando provecho del Zipper

Abstrayendo la modificación

- Modificación generalizada del elemento en foco

```
modify :: (a -> a) -> Zipper a -> Zipper a
modify f (Branch x l r, bs) = (Branch (f x) l r, bs)
modify f (Leaf, bs)         = (Leaf, bs)
```

con la intención de escribir cosas como

```
> let newFocus1 = modify (\_ -> 'N')
                  (goLeft (goLeft (testTree,[])))
> let newFocus2 = modify (\_ -> 'e')
                  (goBack newFocus1)
```

Ahora la modificación es independiente del movimiento

UNIVERSIDAD SIMÓN BOLÍVAR

Sacando provecho del zipper

Aumentando la estructura

```
attach :: Tree a -> Zipper a -> Zipper a
attach t (_,bs) = (t,bs)
```

UNIVERSIDAD SIMÓN BOLÍVAR

Sacando provecho del zipper

Aumentando la estructura

```
attach :: Tree a -> Zipper a -> Zipper a
attach t (_,bs) = (t,bs)
```

- Si el foco era un nodo interno, `attach` nos permite reemplazarlo por un árbol nuevo o podarlo poniendo una hoja.

UNIVERSIDAD SIMÓN BOLÍVAR

Sacando provecho del zipper

Aumentando la estructura

```
attach :: Tree a -> Zipper a -> Zipper a
attach t (_,bs) = (t,bs)
```

- Si el foco era un nodo interno, `attach` nos permite reemplazarlo por un árbol nuevo o podarlo poniendo una hoja.
- Si el foco era una hoja, `attach` nos permite reemplazarlo por un árbol nuevo para extenderlo.

¡Manteniendo el foco para continuar transformando!

Sacando provecho del zipper

It's a long way to the top if you want to rock'n roll

```
tothetop :: Zipper a -> Zipper a
tothetop (t,[]) = (t,[])
tothetop z      = tothetop $ goBack z
```

Sacando provecho del zipper

It's a long way to the top if you want to rock'n roll

```
tothetop :: Zipper a -> Zipper a
tothetop (t,[]) = (t,[])
tothetop z      = tothetop $ goBack z
```

- Así que, comenzamos por enfocarnos

```
focus :: Tree a -> Zipper a
focus t = (t,[])
```

- Movemos el foco a placer usando `goRight`, `goLeft` y `goBack`.
- Quizás haciendo cambios con `modify` en nuestro paseo.
- Regresamos al tope con `tothetop` y recuperamos el árbol cambiado

```
defocus :: Zipper a -> Tree a
defocus (t,_) = t
```

Manteniendo el foco con seguridad

I just moved. So here's my focus, maybe

```
goLeft :: Zipper a -> Maybe (Zipper a)
goLeft (Branch x l r,bs) = Just (l,LeftCrumb x r:bs)
goLeft (Leaf,         _) = Nothing

goRight :: Zipper a -> Maybe (Zipper a)
goRight (Branch x l r,bs) = Just (r,RightCrumb x l:bs)
goRight (Leaf,         _) = Nothing

goBack :: Zipper a -> Maybe (Zipper a)
goBack (t, LeftCrumb  x r:bs) = Just (Branch x t r,bs)
goBack (t, RightCrumb x l:bs) = Just (Branch x l t,bs)
goBack (t, [])                = Nothing
```

- Queremos impedir que el foco se caiga del árbol.
  - goLeft y goRight no pueden procesar hojas.
  - goBack no puede pasar de la raíz del árbol.

El zipper para listas es sencillo

Porque sólo hay dos direcciones

- Definimos el foco y la historia

```
type ListZipper a = ([a],[a])
```

El zipper para listas es sencillo

Porque sólo hay dos direcciones

- Definimos el foco y la historia

```
type ListZipper a = ([a],[a])
```

- Los movimientos de avance y retroceso

```
goForward :: ListZipper a -> ListZipper a
goForward (x:xs, bs) = (xs,x:bs)

goBack :: ListZipper a -> ListZipper a
goBack (xs, b:bs) = (b:xs, bs)
```

El zipper para listas es sencillo

Porque sólo hay dos direcciones

- Definimos el foco y la historia

```
type ListZipper a = ([a],[a])
```

- Los movimientos de avance y retroceso

```
goForward :: ListZipper a -> ListZipper a
goForward (x:xs, bs) = (xs,x:bs)

goBack :: ListZipper a -> ListZipper a
goBack (xs, b:bs) = (b:xs, bs)
```

- Y el retorno al inicio

```
toFront :: ListZipper a -> ListZipper a
toFront (l,[])      = (l,[])
toFront (xs, b:bs) = toFront (b:xs, bs)
```

Parece una cremallera en movimiento. . . o no.

Recorriendo un Laberinto

Parece que Teseo sabía Haskell...

- Modelaremos un laberinto como un árbol de puntos de decisión

```
data Node a = DeadEnd a
            | Passage a (Node a)
            | Fork    a (Node a) (Node a)
            deriving (Eq, Show)
```

  - Un "pasillo ciego".
  - Un pasillo que conecta con otro punto de decisión.
  - Una bifurcación en la cual hay otros dos pasillos, además de aquel por el cual llegamos – podría ser una lista, pero vamos a mantenerlo simple.
- El contenido permitiría tener coordenadas, tesoros, monstruos, princesas o todas las anteriores...

Recorriendo el laberinto

El hilo de Ariadna

- Definimos el tipo para dejar el rastro

```
data Branch a = StraightOn a
              | TurnLeft  a (Node a)
              | TurnRight a (Node a)
              deriving (Eq, Show)
type Thread a = [Branch a]
```

  - En un "pasillo ciego" no hay nada que hacer – no hay acción.
  - En un pasillo simple, se puede avanzar.
  - En una bifurcación, se puede tomar la izquierda o la derecha – recordando el pasillo que no tomamos.
  - El hilo de Ariadna es una secuencia de esas decisiones.
- En todas las acciones conservamos el contenido del nodo particular.

Recorriendo el laberinto

Foco + rastro = zipper

- Ya podemos definir el zipper para este laberinto

```
type Zipper a = (Node a, Thread a)
```

- Foco – parte “no explorada” del laberinto en este paseo.
- Rastro – cómo regresar sobre nuestros pasos.

Recorriendo el laberinto

Tomar decisiones y avanzar

- Implantamos la funcionalidad para avanzar

```
left :: Zipper a -> Maybe (Zipper a)
left (Fork x l r,t) = Just (l,TurnLeft x r:t)
left _              = Nothing

right :: Zipper a -> Maybe (Zipper a)
right (Fork x l r,t) = Just (r,TurnRight x l:t)
right _              = Nothing

straight :: Zipper a -> Maybe (Zipper a)
straight (Passage x n,t) = Just (n,StraightOn x:t)
straight _               = Nothing
```

- No tiene sentido girar en un pasillo normal ni en uno ciego.
- No tiene sentido avanzar en un pasillo ciego ni una bifurcación.

UNIVERSIDAD SIMÓN BOLÍVAR

Recorriendo el laberinto

Regresar, quizás hasta el principio

- Implantamos la funcionalidad para regresar

```
back :: Zipper a -> Maybe (Zipper a)
back (_,[])                = Nothing
back (n, StraightOn x:t)   = Just (Passage x n, t)
back (l, TurnLeft x r:t)   = Just (Fork x l r,  t)
back (r, TurnRight x l:t)  = Just (Fork x l r,  t)

entrance :: Zipper a -> Maybe (Zipper a)
entrance (n,[]) = Just (n,[])
entrance z      = entrance $ fromJust $ back z
```

- Y ya tenemos toda la maquinaria para recorrer el laberinto.

Escribiendo Zippers

- Para construir un zipper es necesario contemplar:
  - Cuántas "direcciones" de movimiento hay en la estructura – en la lista hay dos, en los árboles tres, puede haber más...
  - Cuántas transformaciones son posibles según los contenidos
    - Listas y árboles de ejemplo tienen *un* sólo tipo a.
    - Si hay más de un tipo contenido, en ocasiones puede hacerse un sólo zipper "combinado" pero también pueden ser necesarios varios zippers que se usan por separado.
- GHC aún no puede derivar un Zipper automáticamente.

Derivando zippers. . . literalmente

Here be math

- Un tipo de datos de la forma

```
data This = Foo | Bar | Baz
```

se conoce como **tipo suma** y se puede escribir

$$This = Foo + Bar + Baz$$

Derivando zippers. . . literalmente

Here be math

- Un tipo de datos de la forma

```
data This = Foo | Bar | Baz
```

se conoce como **tipo suma** y se puede escribir

$$This = Foo + Bar + Baz$$

- Un tipo de datos de la forma

```
data This = This Foo Bar Baz
```

se conoce como **tipo producto** y se puede escribir

$$This = Foo \times Bar \times Baz$$

UNIVERSIDAD SIMÓN BOLÍVAR

Derivando zippers...literalmente

Here be more math

- Un tipo recursivo, polimórfico y combinado como

```
data Tree a = Leaf | Node a (Tree a) (Tree a)
```

  es más complejo de representar.
  - Para comenzar es un tipo suma, con dos constructores.
  - El constructor `Leaf` no es polimórfico.
  - El constructor `Node` es polimórfico y tipo producto.

UNIVERSIDAD SIMÓN BOLÍVAR

Derivando zippers. . . literalmente

Here be more math

- Un tipo recursivo, polimórfico y combinado como

```
data Tree a = Leaf | Node a (Tree a) (Tree a)
```

  es más complejo de representar.
  - Para comenzar es un tipo suma, con dos constructores.
  - El constructor `Leaf` no es polimórfico.
  - El constructor `Node` es polimórfico y tipo producto.
- La historia corta, es que se escribe

$$Tree\ a = 1 + a \times (Tree\ a) \times (Tree\ a)$$

Derivando zippers. . . literalmente

Here be even more math

- El zipper para el árbol binario tiene el tipo

```
data Crumb a = LeftCrumb a (Tree a)
             | RightCrumb a (Tree a)
```

UNIVERSIDAD SIMÓN BOLÍVAR

Derivando zippers... literalmente

Here be even more math

- El zipper para el árbol binario tiene el tipo

```
data Crumb a = LeftCrumb a (Tree a)
             | RightCrumb a (Tree a)
```

- Con ecuación...

$$Crumb\ a = a \times (Tree\ a) + a \times (Tree\ a)$$

- ...que al combinar términos similares queda

$$Crumb\ a = 2 \times a \times Tree\ a$$

UNIVERSIDAD SIMÓN BOLÍVAR

Derivando zippers. . . literalmente

Here be perturbing math

- El tipo de datos original tiene ecuación

$$Tree\ a = 1 + a \times (Tree\ a) \times (Tree\ a)$$

- El tipo de datos para el zipper que usamos es

$$Crumb\ a = 2 \times a \times (Tree\ a)$$

Derivando zippers. . . literalmente

Here be perturbing math

- El tipo de datos original tiene ecuación

$$Tree\ a = 1 + a \times (Tree\ a) \times (Tree\ a)$$

- El tipo de datos para el zipper que usamos es

$$Crumb\ a = 2 \times a \times (Tree\ a)$$

$$\partial_{Tree\ a}(1 + a \times (Tree\ a) \times (Tree\ a)) = 2 \times a \times (Tree\ a)$$

Derivadas de tipos, en serio. . .
Para calcular el tipo óptimo para el rastro.

UNIVERSIDAD SIMÓN BOLÍVAR

Derivando zippers. . . literalmente

Here be the next to last math

- Para el caso del laberinto

```
data Node a = DeadEnd a
            | Passage a (Node a)
            | Fork    a (Node a) (Node a)
```

- Tiene ecuación

$$Node\ a = a + a \times (Node\ a) + a \times (Node\ a) \times (Node\ a)$$

UNIVERSIDAD SIMÓN BOLÍVAR

Derivando zippers. . . literalmente

Here be the last math

- Derivamos

$$\partial_{Node\ a}(a+a\times(Node\ a)+a\times(Node\ a)\times(Node\ a)) = a+2\times a\times(Node\ a)$$

- Expandimos a sumas sin constantes

$$a + a \times (Node\ a) + a \times (Node\ a)$$

- Así que el rastro tiene que tener tipo

```
data R a = Foo a
         | Bar a (Node a)
         | Baz a (Node a)
```

Uno escoge los nombres de los constructores.

UNIVERSIDAD SIMÓN BOLÍVAR

Zippers en la práctica

¿Qué hago si necesito un zipper?

- Trabajar con árboles en Haskell implica usar `Data.Tree`
  - Implanta árboles generalizados – Rose Trees.
  - `Data.Tree.Zipper` – `rosezipper` en Hackage.
- Para tipos recursivos arbitrarios, es necesario construir el zipper manualmente con el método estudiado hoy.

Quiero saber más. . .

- Functional Pearl: The Zipper
  Huet
- The Derivative of a Regular Type is its Type of One-Hole Contexts
  Connor McBride